Opportunity Cost of

Technical Debt

Minimize Your
Rich Text Editor
Development

1.0

Introduction

All Successful Technology has Debt

As they say, money talks.

So by saying you have technical debt, you've achieved an implied level of success.

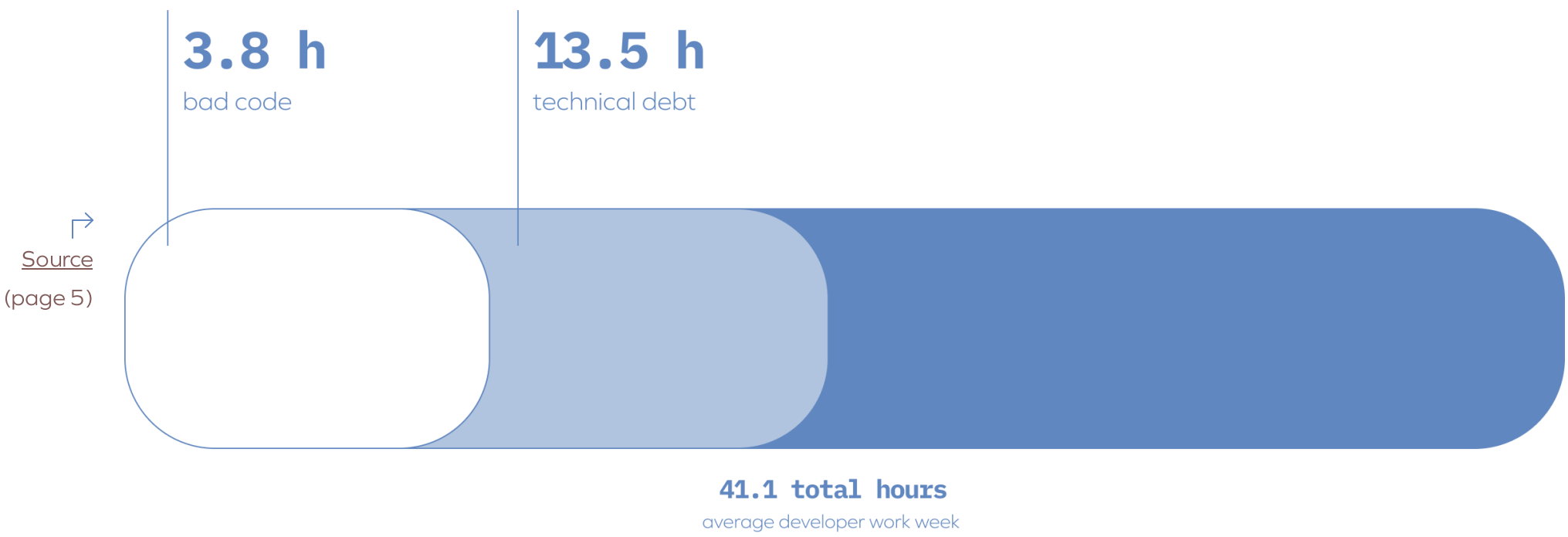
It means the product in question is being bought, used, and requires rapid change to support revenue hypergrowth, user demands or product-market fit. But, there's a sticky trap: innovation and development go hand-in-hand with technical debt.

In the pursuit of delivering more, faster, your development teams and infrastructure are taking on tech debt. And, the time soon arrives where that debt outranks your growth projects; it begins slowing development and delaying new releases.

Every bug and payoff task carries an opportunity cost.

Whenever your Dev team spends time and resources on those payoff tasks (upgrading, refactoring, repairing), they're pursuing fewer software improvements and innovations. The potential monetary value yielded from those innovations, is forfeited.

According to Stripe's Developer Coefficient report, 42% of every developers' working week is spent dealing with technical debt (13.5 hrs) and bad code (3.8 hrs), which equates to nearly $85 billion worldwide in opportunity cost lost annually. That's a huge chunk of development time and money.



The report goes on to say that with "…the number of developers increasing year-over-year at most companies, developers working on the right things can accelerate a company's move into new markets or product areas and help companies differentiate themselves at disproportionate rates."

Perhaps then, the post-pandemic software challenge isn't more new products and features… it's the ability to walk the tightrope between technical debt and innovation. Too much debt, sucks energy away from innovation and slows growth. Opportunities are lost. Too little debt, and your ability to scale fast, ship fast, and test in-market can be stifled. Value to your users slows.

Kelly Sutton
Software Engineer

> In conversations and writing around technical debt, folks stretch the metaphor. We dive deep into the finance comparisons, we analyze its softer costs, or we just throw our hands up and claim it doesn't exist. […] To get others in your organization, whether it be your manager or her manager or someone in finance, you need to have a common language.
>
> This becomes especially important when discussing opportunity cost. How do you rank technical debt projects against growth projects? At a certain point, we need to quantify "Slowing down development time."

Wherever technical debt lives, decisions are framed by that debt.

Stakeholder debates rage over whether to manage the debt or become a dead company with pristine code. Meetings echo with refrains like "Yes, we should do something about the tech debt, but we can't afford to do it now". Then there's the development work you've taken on that's beyond your team's core skill set — building complex components that carry greater risk of build blowouts and debt accrual — that have mushroomed your debt levels.

What's the answer?

Smart organizations are finding ways to sidestep making their debt hole deeper. Their tech debt management plan includes scoping, sourcing and maintaining a reusable tech stack that includes components with built-in scalability.

They're minimizing their 'owned' tech debt by buying, assembling and integrating those complex components, from specialists.

So someone else carries the tech debt load.

Tech Debt Defined

According to Techopedia, "Technical debt is a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution."

2.0

The Speed of Change

2.1

Does Being Agile Increase Tech Debt?

Technical debt is hard to avoid.

It's part of the air that every software developer breathes.

As McKinsey says, "Technical debt is like dark matter: you know it exists, you can infer its impact, but you can't see or measure it. Product delays, hidden risks, spiraling costs, and even engineers leaving in frustration are all common symptoms."

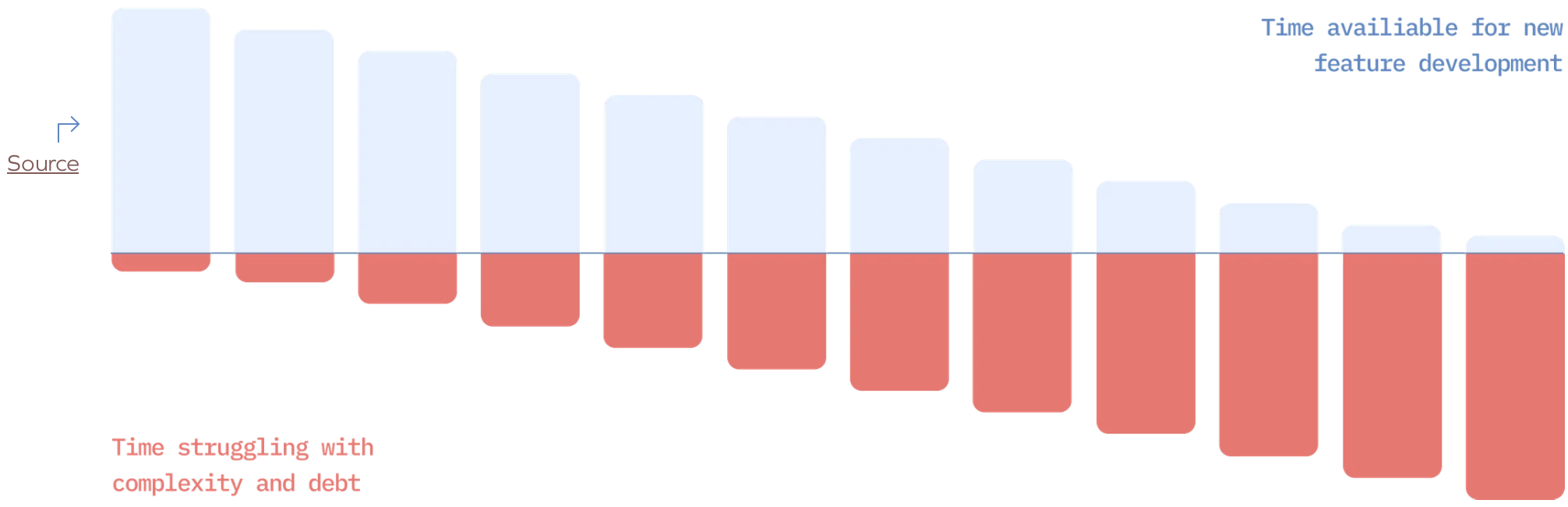
Another characterization — the 'off-balance-sheet' accumulation of future development work — doesn't sound so bad, but it's almost invisible bloat (especially to non-technical stakeholders) can cripple your long-term ability to deliver value to users.

Tech debt is a ceaseless, clingy cycle.

As software proliferates, enterprises push harder towards digital transformation. They jostle to increase their agility, keep pace with software-based innovation and customer expectations. That forces accelerated development timeframes and debt accumulation.

Agile product and engineering teams struggle to meet the demands of modern architectures and edge cases (especially those outside their core skill), and cycle through hit-and-miss release schedules at breakneck speed to build secure, reliable applications.

Technical debt and the consequences on feature development



Let's attach some hard numbers to the issue.

$1.31T

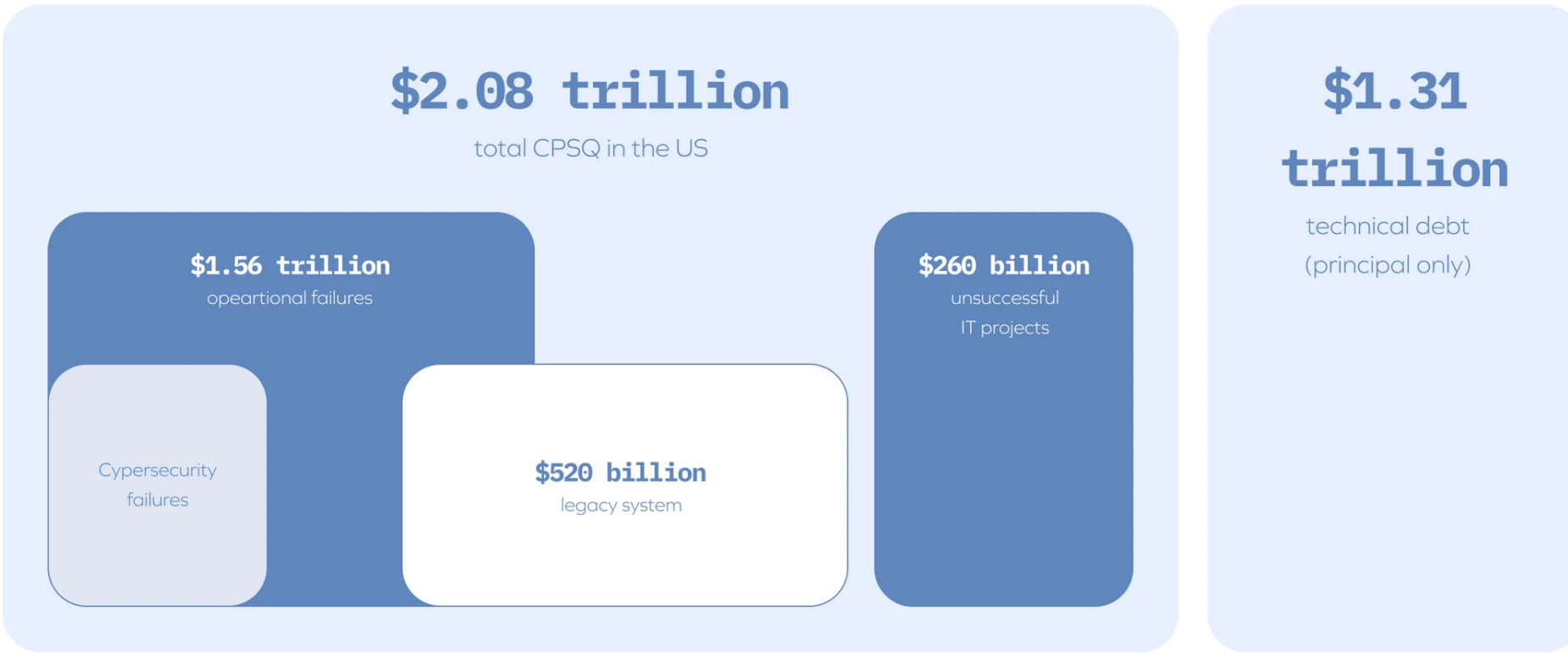
estimated software technical debt (principal only, not including interest) in 2020

A 2020 report from the Consortium for Information & Software Quality (CISQ) calculated the Total Cost of Poor Software Quality (CPSQ) in the United States to be $2.08 trillion (T). For comparison, only a dozen countries have an annual GDP of $2 trillion or more.

The same report noted that the 2020 estimated software technical debt was $1.31 trillion (principal only, not including interest), as an additional future cost. Those costs have been increasing at a rate of 14% since 2018.

Source
(Page 4)

Total Cost of Poor Software Quality (CPSQ) in the United States in 2020



The magnitude of the issue feels worse, when it's in your own sandbox.

A McKinsey study in 2022 reported that "Some 30 percent of CIOs we surveyed believe that more than 20 percent of their technical budget ostensibly dedicated to new products, is diverted to resolving issues related to tech debt. Furthermore, they estimate that tech debt amounts to 20 to 40 percent of the value of their entire technology estate (before depreciation)."

Tech debt is

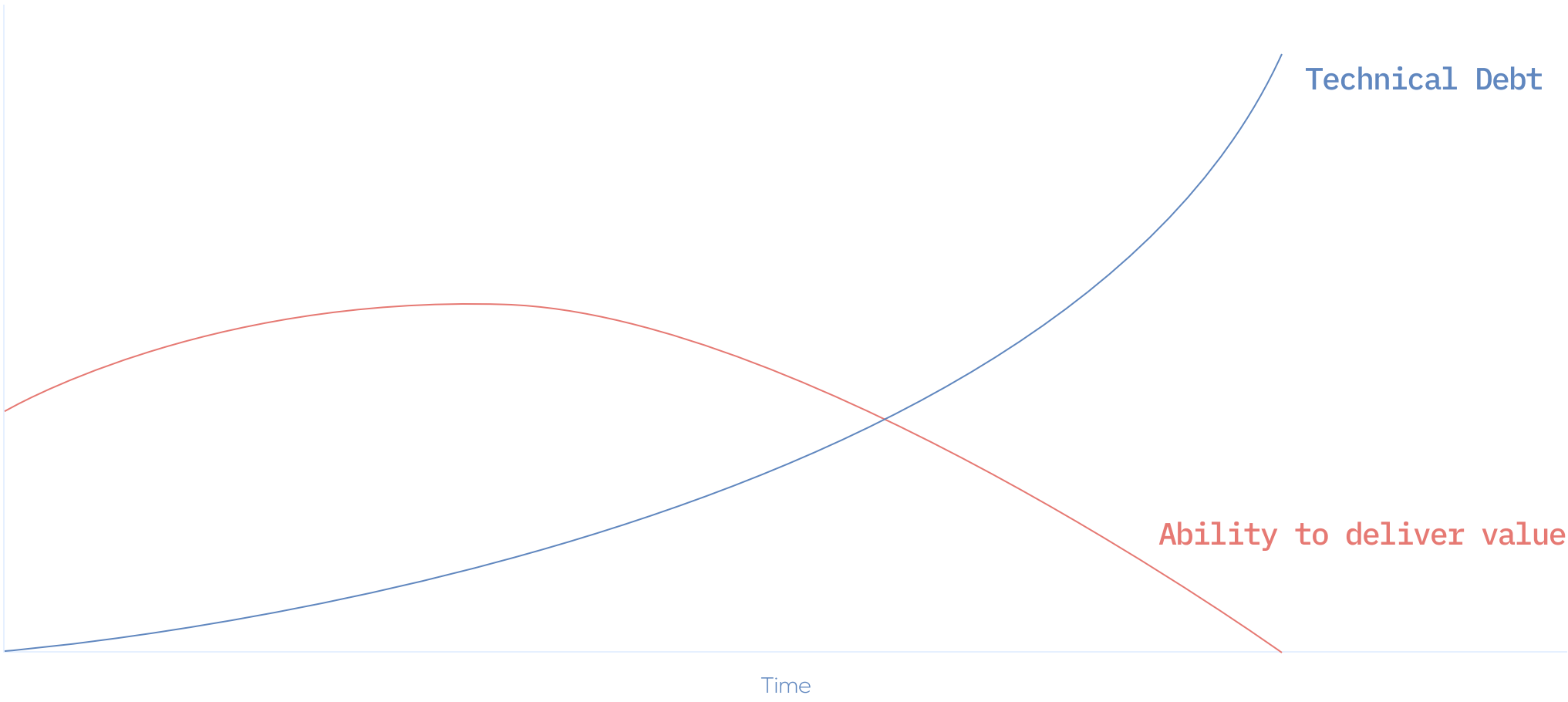
20-40%

of the total value of their entire technology estate

When more and more is spent on refactoring, it slows product innovation. And it lowers morale. That's not what your Dev team signed up for — they want to work on meaningful projects.

As a long-time product leader, David Pereira recognised the issue in a recent article, "Ignoring tech debt is the right way of building a product no developer wants to work with. […] Building a scalable product requires a strategy. It's impossible to keep a product maintainable if the only thing you do is adding more features to it. Yet, it doesn't mean you should build everything scalable from the beginning."

The trick is knowing what to build, what to buy and what specialists to tap.



Taking on complex, costly code work that's outside your dev team's strategic focus, often produces shortcuts, low-quality, temporary-but-not-really workarounds and other hackey short-term solutions… to get through a launch window.

But they're not viable long-term.

McKinsey noted in their CIO interviews on tech debt, that "Some companies find that actively managing their tech debt frees up engineers to spend up to 50 percent more of their time on work that supports business goals. The CIO of a leading cloud provider told us, 'By reinventing our debt management, we went from 75% of engineer time paying the [tech debt] 'tax' to 25%. It allowed us to be who we are today.'"

Smart dev teams are reinventing their debt management approach.

In doing so they reduce risk, lift morale and deliver positive value. They avoid taking on the unnecessarily complex development work (and its tech debt baggage) and instead buy and integrate reusable components, from specialists.

Assembling specialist components minimizes the 'owned' tech debt they need to carry. The debt that 'would have' been generated by building those complex components, is measurably reduced (if not almost eliminated).

In its place, the potential gain from on-time launches of new features is realized.

Who Invented the Term Technical Debt?

Ward Cunningham, the developer of the first wiki and co-author of the Agile Manifesto, first explained and aligned technical debt, using the metaphor of financial debt.
The *principal* part of your technical debt is the time saved during initial implementation, while the *interest* is the additional time, quality, and risk costs incurred until it's resolved.

Does it work? Yes. Assembling maximizes the horsepower behind third-party specialization, empowers developers to focus on what they do best, minimizes needless technical debt and delivers the nimbleness to answer market demands for innovation.

Renowned software developer, author and speaker, Martin Fowler nails it when he says: "[…] while you're programming, you are learning. It's often the case that it can take a year of programming on a project before you understand what the best design approach should have been. […]

That's a crucial part of reinventing your tech debt management. Don't needlessly expand your own tech debt through complex builds that other specialists have already perfected.

Let the specialists manage the bear traps.

Open Source: a Plus or Minus for Tech Debt?

Old questions still have echoes.

Why would I use open source?

Here's the new answer: You likely already do.

Today, it's everywhere. Companies of all sizes — from small Dev teams to large-scale enterprises — use it frequently. Why? Because when you're trying to innovate, fast, open source software (OSS) gives you a head start… it always has done.

Open innovation isn't new. The first "Oxford English Dictionary" was an open-source project, and in the 1950s software was distributed free of charge.[1]

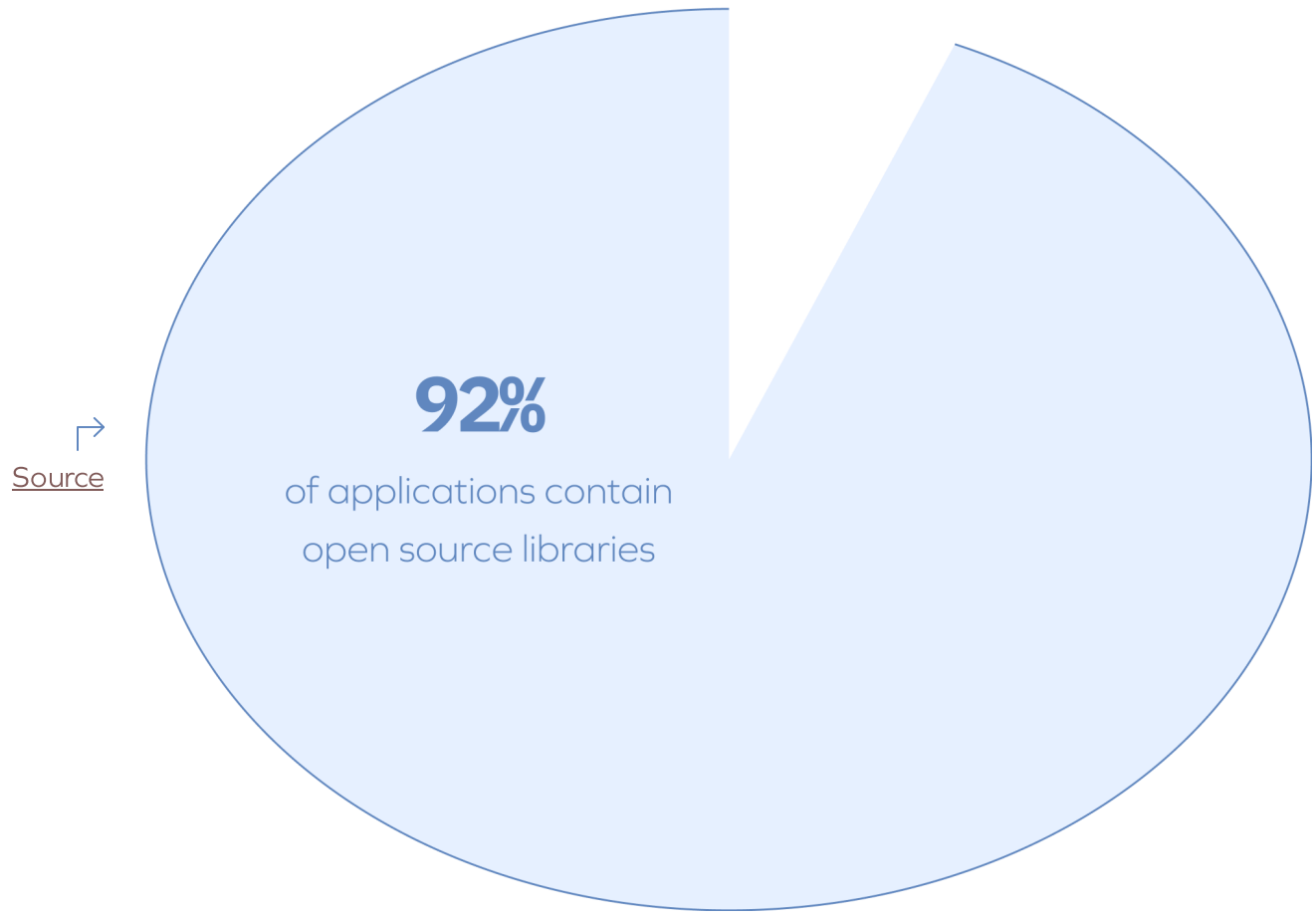
Fast forward several decades and with the emergence of the tech assembly approach called 'digital factory', open source components are increasingly being used to facilitate faster project completions for both open and proprietary applications. They're the outlier X-factor that, used cleverly, can positively impact growth and digital transformation.

The key is knowing which ones to deploy, in wise ways.

Even McKinsey agrees. During the development of their Developer Velocity Index (DVI) they said, "… across the entire group of companies surveyed, a different driver emerged as the biggest differentiator for companies within the top quartile: open-source adoption."

What's a digital factory?

Research firm, McKinsey, has dubbed a factory-like tech assembly approach, a 'digital factory' – where a company "brings together the skills, processes, and inputs required to produce high-quality outputs. […] The best digital factories can put a new product or customer experience into production in as little as ten weeks. The innovation can then be introduced and scaled up across the business in eight to 12 months."



Source

Open source is frequently the foundation of innovation across the software industry.

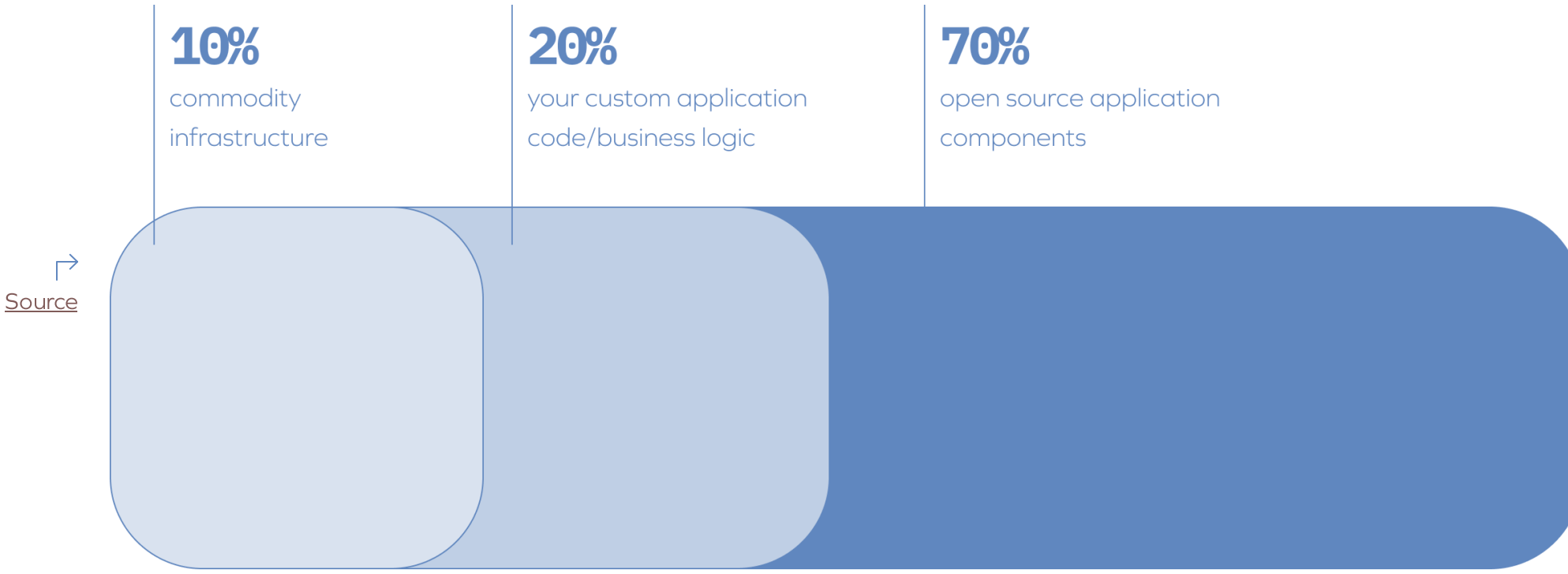
Open source code makes up

70%

of most modern applications

Having transformed from its once heretical roots, it's now an essential factor in how enterprises evaluate, use and purchase software to drive their innovation plans. The rationale is simple: OSS lowers development costs, decreases time to market, increases developer productivity, and accelerates innovation.

Billions of lines of code are freely available and shared through a community of creators, collaborators, and maintainers. As a result, most modern applications are built using open source components — in many cases, open source makes up more than 70% of the code.



Source

Digging deeper, the reasoning for open source use is not dissimilar to other third-party SaaS or software purchases. Open source lets you leverage the velocity of your dev team talent — by letting them work on company differentiation instead of projects outside your core strategic focus.

Why reinvent the wheel, when others have already perfected it?

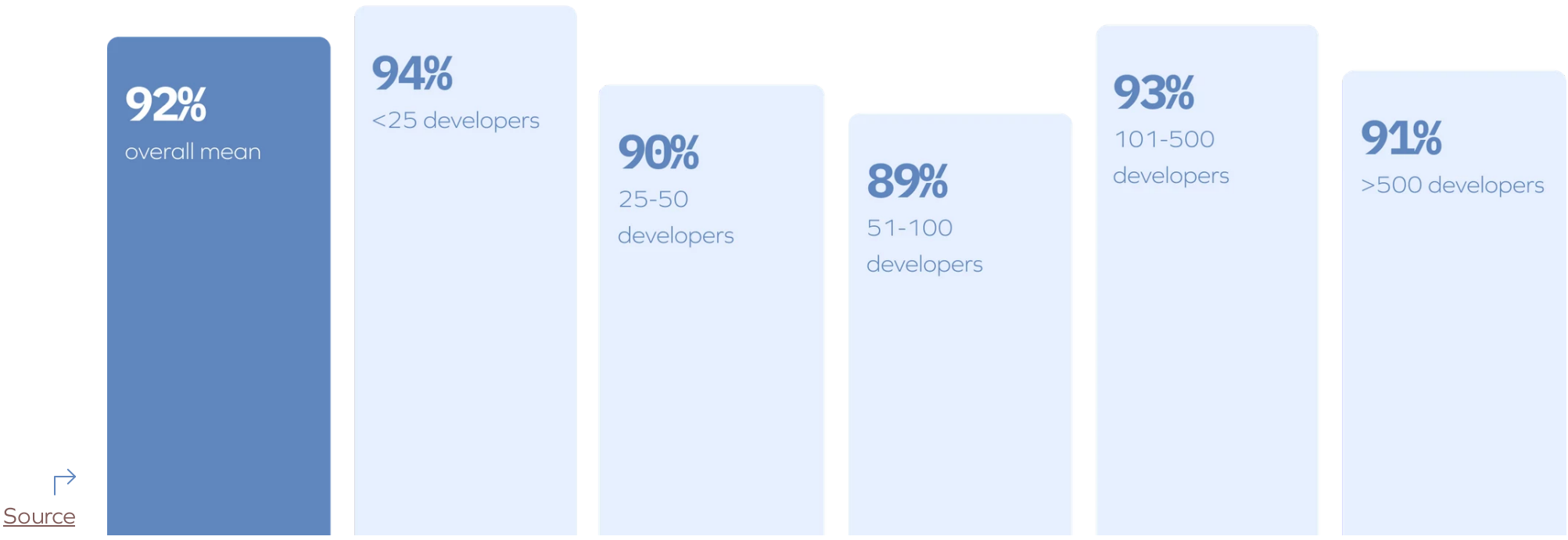
With a 'digital factory' approach, you assemble an agile software stack that comprises both open and closed components, which are curated, vetted, and professionally managed. It saves time, money and boosts your speed-to-market.

The quality of open source code has certainly improved. In part, that's due to the large software companies now contributing to Github, many of whom use an open core business model, apply OSS development methodologies, and use established Quality Control (QC) processes.

In 2018, a Tidelift Open Source Survey highlighted that companies of all sizes use open source components in 92% of their project applications. Virtually all the organizations surveyed said they use OSS to take advantage of increased productivity, reduced cost and time to deployment.

It's obvious that open source continues to drive innovation. At speed.

Average percentage of projects using open source by company development team size



Source

But there's a downside. The push for faster product launches and redirecting your developers' time also forces an open-eyed view of the current situation.

—
The Tidelift guide to securing your open source dependencies

Let's consider just a few of the ways your team wastes time managing your open source dependencies rather than developing your product:

Staying up to date with the latest bugfix versions.

Moving to a new major version of a framework or library.

Dealing with bugs or security issues related to an unmaintained dependency.

Handling requests from your legal department to list every package you're using, along with their licenses.

Documenting everything you use for your security team and addressing live vulnerabilities.

Being honest, that situation isn't ideal. But commercial development teams rely on open source to innovate fast.

Then there's the tech debt burden that open source can generate. In most closed software, updates are automatically *pushed* to users. Conversely, open source uses a *pull* model where users are responsible for updating. If you fail to deploy the necessary major, minor or patch updates, you expose yourself to more hidden tech debt.

What's Managed Open Source Software?

Managed open source software and components, maximize the speed-to-market, while minimizing potential risks to users. All license details, security risks and maintenance issues are clearly communicated (if they occur) and upgrades are regularly released.

What's the answer? Using professionally managed open source components.

According to Tidelift, managed open source "… allows application development teams to speed up development and reduce risk by outsourcing the management of open source components to the experts who create and maintain them."

With a clear process for vetting (Software Bill of Materials, SBoM), monitoring (automated visibility and control of the OSS) and reusing (cataloging) open source components, enterprises can reuse OSS and third-party specialist components to rapidly jump-start their innovation programs.

Managed open source components minimize your tech debt burden.

And are a cheat sheet to your success.

What's a Software Bill of Materials (SBoM)?

According to Gartner, "SBoMs are an essential tool in your security and compliance toolbox. They help continuously verify software integrity and alert stakeholders to security vulnerabilities and policy violations." They are an increasingly common and critical component of software development lifecycle (SDLC) and DevSecOps processes.

In 2021 several high-profile security breaches prompted the US President to issue an Executive Order on Cybersecurity, (May 2021). It included a requirement that software vendors provide a SBoM (Software Bill of Materials) for those selling to the US Federal Government. The order includes a provision that will require IT vendors to provide an SBOM with software and hardware.

2.3

Complexity Breeds Constant Reinvention

How does complexity play with technical debt?

Rich text editor (RTE) components look simple to build.

In reality, they're one of the most complex interface components of a tech stack: visually and technically. Not unexpectedly, their simple exterior belies the intricate complexities inside.

'ENTER' KEY PERFORMS OVER 100 DIFFEREN T BASE ACTIONS

First, let's cover the basics. A rich text editor is an interface for editing 'rich text'. It's a crucial component within every organization's tech stack that enables rich text editing capabilities within any application — no matter the type, use case, or device. Now the nitty gritty.

The basic 'Bold' text button can have 40+ different calls and interactions.

The ubiquitous 'Enter' key performs over 100 different base actions — depending on where you are in the editor interface and what you're doing. That excludes browser compatibility issues or dependencies built into the product, so the final tally is likely double the number of base actions it performs.

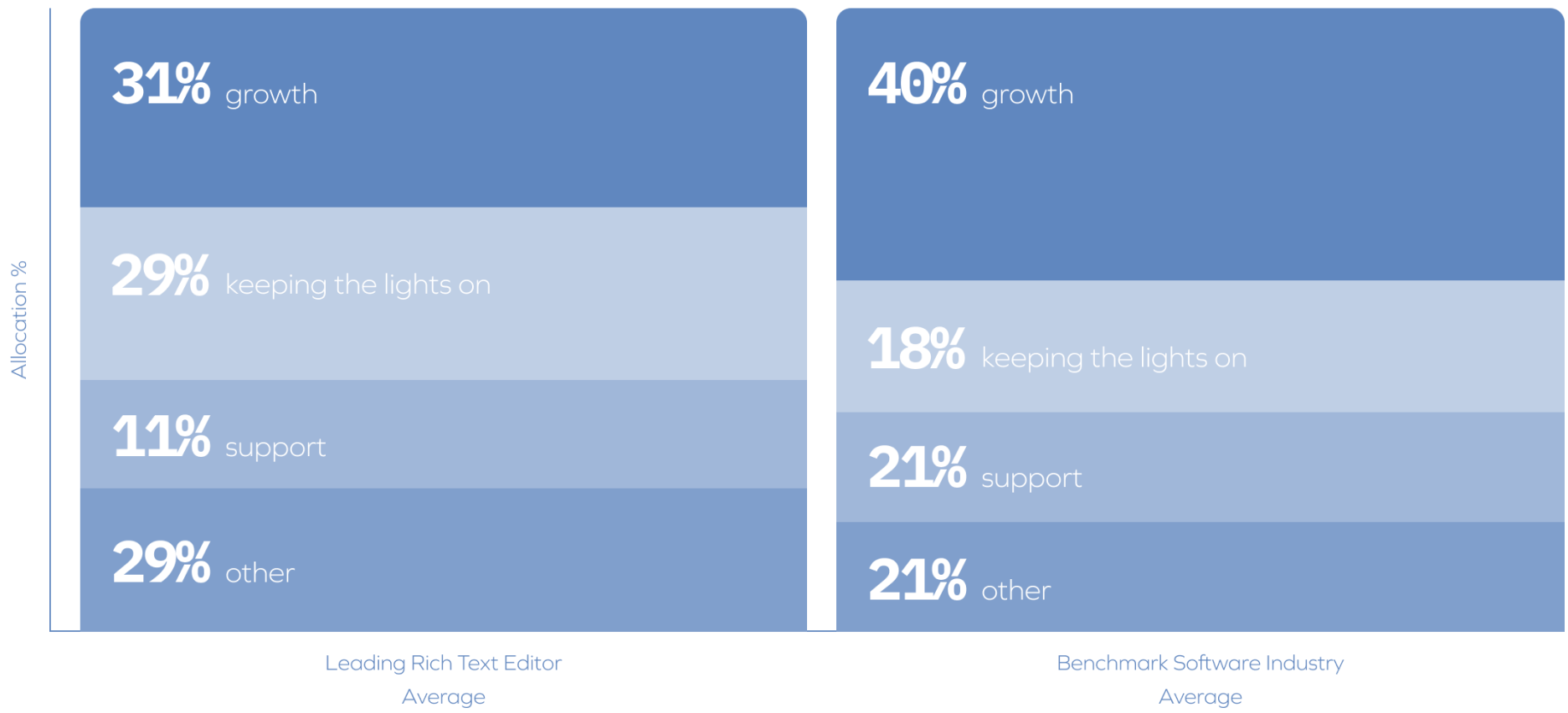
It's probably obvious by now that rich text editor development requires deep domain knowledge.

However, it's not only the main editor's initial coding and subsequent refactoring that needs to be done. Each individual plugin (feature), every linked service, library, or tool, as well as browser changes and APIs, need to be monitored, reworked, and maintained.

It's a flywheel in constant motion.

Which makes tracking technical debt in a rich text editor — and even more importantly, identifying areas that likely have tech debt — very difficult. Quantifying the volume and value of that technical debt burden is equally hard.

Although, there are indications… but they're not comforting either.



While the yearly average spend on 'growth' (new) activity in the software industry is 40%, for a rich text editor, it's 31%. In the area of technical debt, the industry average spends 18% while a rich text editor carries around 29% ongoing in technical debt. That's a lot of tech debt to take on, if you're *not a domain specialist* who knows the right decisions when it comes to [intentional and unintentional tech debt.]

> Technical Debt is not necessarily a bad thing — as long as there is a real plan to pay it off.
>
> — **Stephanie Ockerman and Simon Reindl** Mastering Professional Scrum

Carried carefully though, a certain level of tech debt isn't a bad thing.

It speeds software development and gets you to a short-term goal — when that deadline is the most important thing. But for RTEs, coding new functionality to match upgrades and features in other applications (and to stop breakages), is an endless cycle.

You need the experience to know when to make the right trade-offs and decisions, to capitalize on an opportunity cost. Because the mismanagement of an RTE's technical debt can be dire.

Most rich text editor specialists have large teams of developers who work on them year-round. And that's all they do. They know *when and why* it's the right decision to take on tech debt and at what point to start paying it down — so that product development and feature improvements don't slow.

Take a look inside the *most basic components* you need to build, test, maintain for a rich text editor, and the ensuing technical debt you need to manage.

Common rich text editor requirements

Core Functionality

These are the no-frills core parts of a basic (minimum expectation) rich text editor

COMPONENT	DESCRIPTION
Contenteditable	Selection
Basic Formatting	Loading Content
HTML Compliance	Input Filtering
Undo	Output Filtering
Focus	Browser Differences

The above requirements cover content and formatting. Next, developers need to look deeper into what content is made of and how to render that within a rich text editing environment.

Additional Functionality

These are not often in a core RTE experience, but are frequently requested by end users

COMPONENT	DESCRIPTION
Links	Images
Embeds	Uploading
Lists	Advanced Formatting
Tables	Emoji

UI Interactions

These UI components affect how users interact and work with the editor

COMPONENT	DESCRIPTION
UI General	UI Dialogs
UI Toolbar	Accessibility
UI Buttons	Touch Devices (mobile/tablets)
UI Menu	Advanced Keyboard Interactions
UI Context Menu	

Look and Feel

How the editor looks and how it's configured to meet product requirements

COMPONENT	DESCRIPTION
Content Editing	Skins and Customizations
Content Published	Configuration

Enterprise Grade Features

These are the advanced features (over and above a core editing experience), that growing SaaS and enterprises demand for their users

COMPONENT	DESCRIPTION
Automated Testing	Spell Checking
Human Testing	Integration
Localization	Performance
Security	

A trusted editing environment requires year-round ongoing maintenance – to keep up with browsers, technologies, changing technologies and how the content is displayed to its ultimate audience, your readers.

> Like financial debt, different companies have different philosophies about the usefulness of debt. Some companies want to avoid taking on any debt at all; others see debt as a useful tool and just want to know how to use debt wisely.

— **Steve McConnell**
Managing Technical Debt White Paper, Construx Software

Rich Text Editors Live a Fast Life

Code doesn't ever really die. But it does decay.

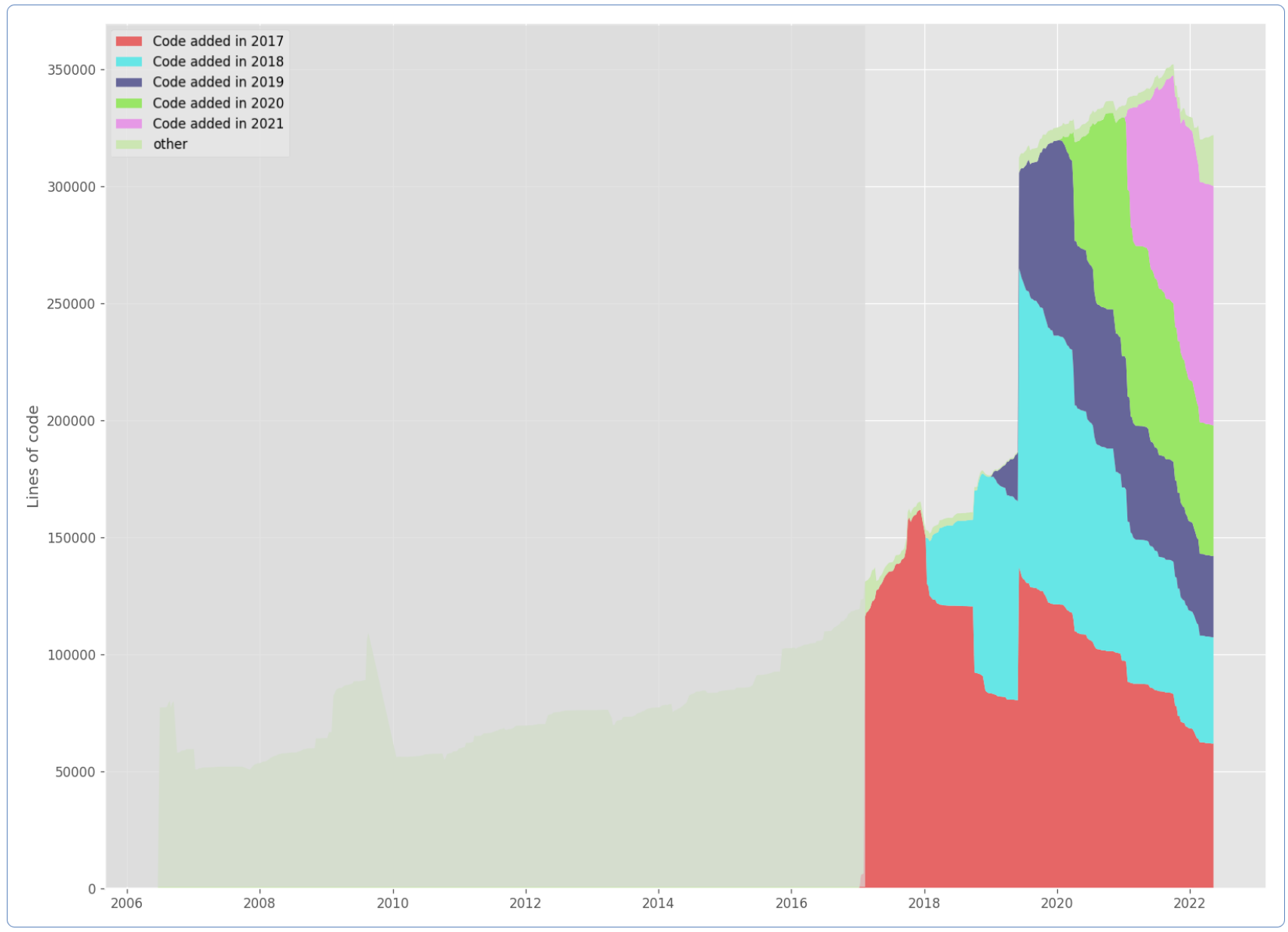
It also has a half-life.

So let's break this idea open. A useful rule of thumb for tech start-ups is to assume all the code you write has a half-life equal to the age of your startup. But that doesn't apply to every piece of software. Or does it?

Paraphrasing Dan North's Software that Fits in Your Head talk, Sandi Metz defines the half-life of code phenomenon as, "the amount of time required for half of an application's code to change so much that it becomes unrecognizable."

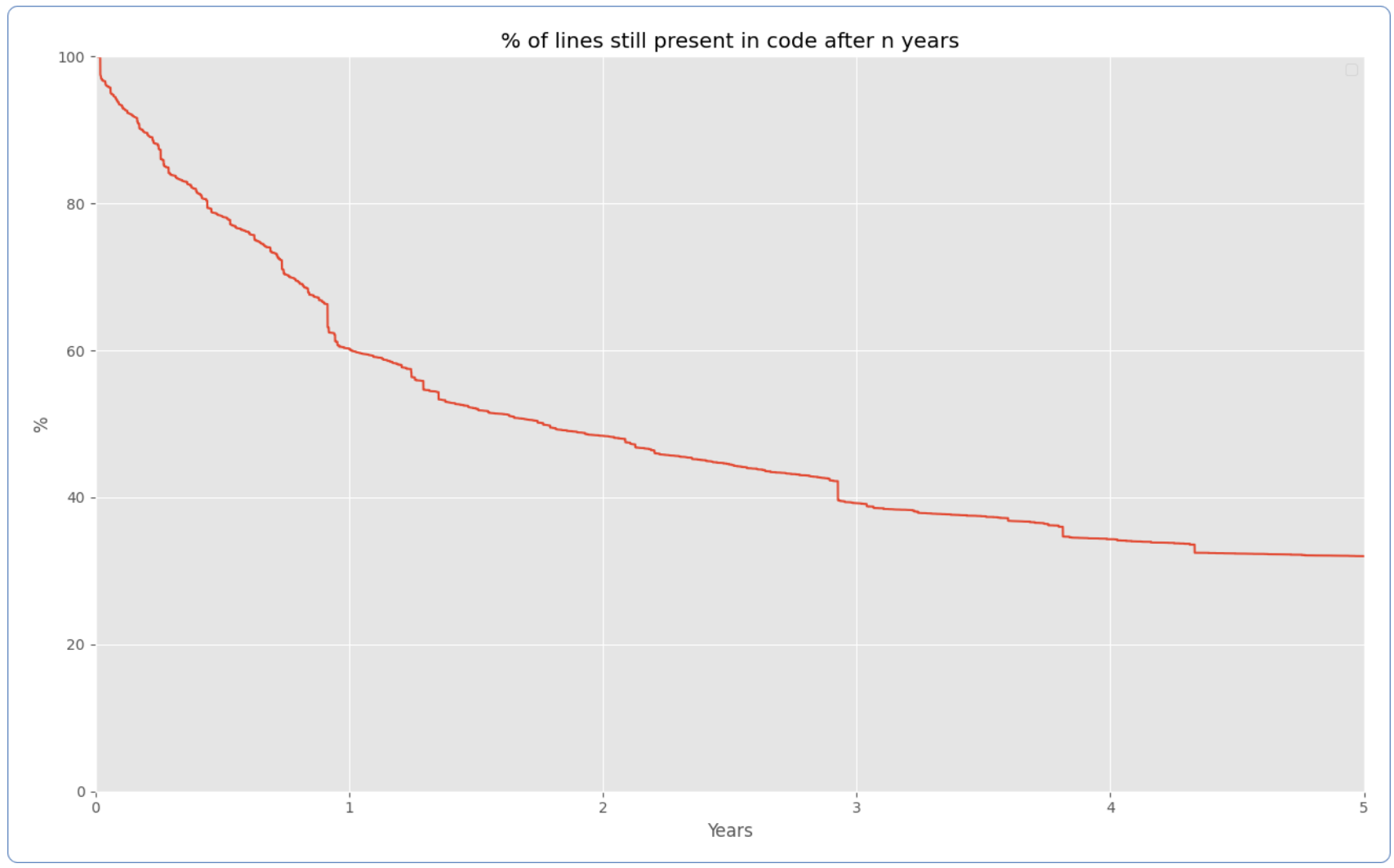
Erik Bernhardsson's 'The half-life of code and the ship of Theseus' article and ensuing Git of Theseus project, explore the conundrum further and question "…does the new code just add up on top of the old code? Or does it replace the old code slowly, over time? […] There is a 'Ship of Theseus' effect, but there's also a compounding effect where codebases keep growing over time."

These articles surface a key aspect of building, maintaining and scaling a rich text editor: its code has a crazy-short half-life. Let's see what that looks like.



The 5-year Half-life of Code in a Rich Text Editor: Lines of code, by yearly cohorts, across 5 years

This first graph (above) plots across 5 years, the **aggregate number of lines of code in a rich text editor**, broken down into cohorts by the year added. It clearly displays the number of sweeping changes needed to maintain the editor's competitive position in the marketplace.



The 5-year Aggregate Decay in a Rich Text Editor: Individual commits in a repo, by yearly cohorts, across 5 years

Next is the **aggregate decay for individual commits in the RTE's core editor repo** (see above). After 5 years, only 30% of the original code remains. The rest has been refactored, upgraded, repaired or replaced.

The editor in question is more than two decades old and has undergone massive resets over that time (switching to TypeScript, rewriting the whole UI and a DOMParser change), yet in the last five years both small incremental changes and large sweeping resets were required to manage the RTE's technical debt. That's a considerable amount of payoff tasks (upgrading, refactoring, repairing) to be done, every year.

Much of the technical debt accrued by a rich text editor, is driven by the code complexity, its dependencies, libraries and browser updates.

Browsers unexpectedly update (see Section 2.5) and open source projects constantly change. Even when no new features are added, a rich text editor must be updated to deal with changes and new feature releases from Chrome, Safari and Firefox.

Otherwise things break.

Of course, the upside of a short software half-life is it's sometimes a great trade-off — because there's often instances when code was designed to be rewritten.

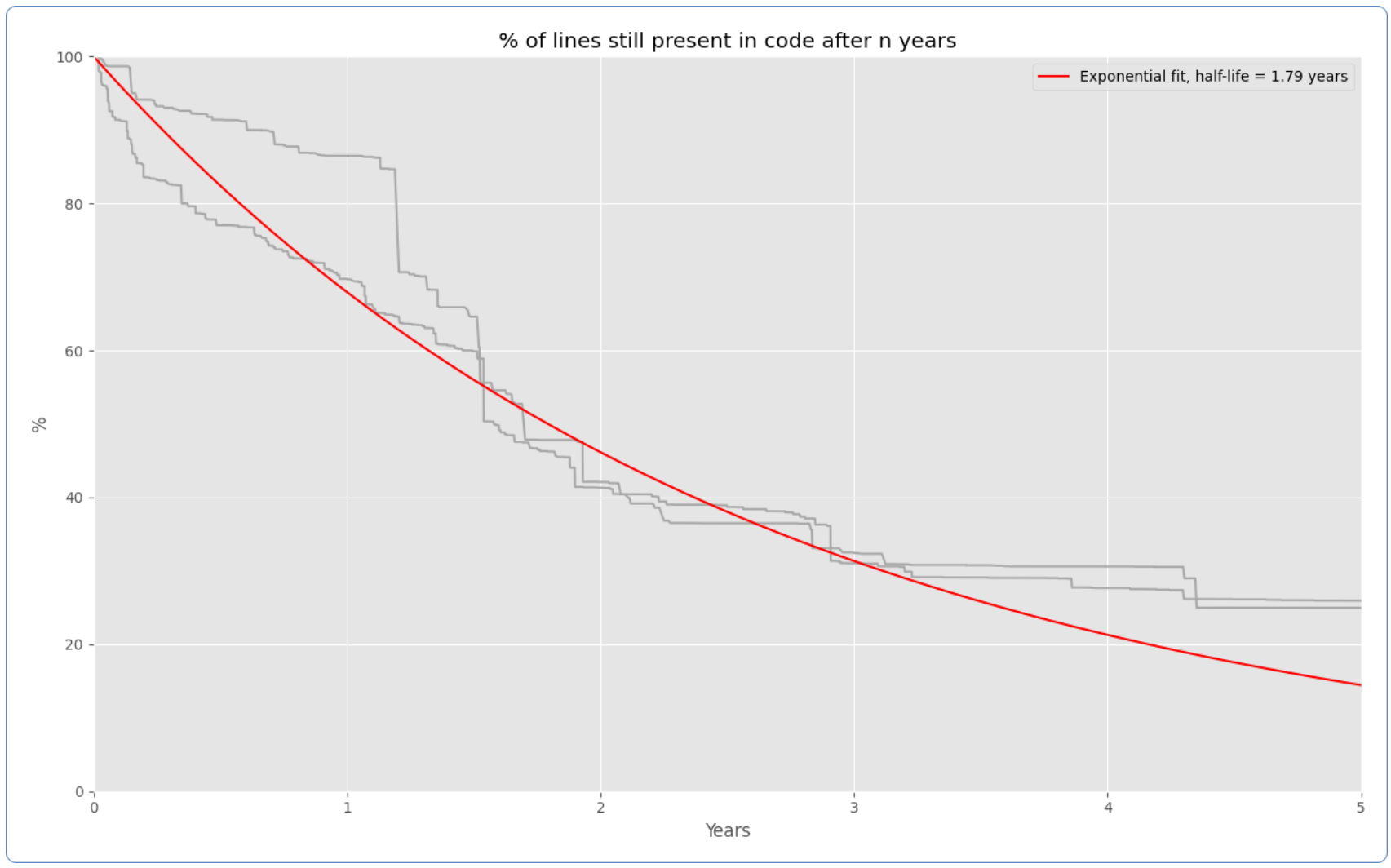
Only

30%

of the original code remains after 5 years

Sandi Metz explains it well in his '[The Half-Life of Code]' post: "The upsides of a short code half-life are significant. Imagine how much better your life would be if your application's code always reflected the most accurate, up-to-date understanding of the problem at hand. Think about how much costs would go down if you never had to navigate dead code. Consider the value of having an application that is free of speculative additions that were thrown in to support features that will never arrive."

For rich text editors, new features do arrive. Frequently. Let's look at what the half-life of the code used in some advanced rich text editor plugins looks like when it's mapped.

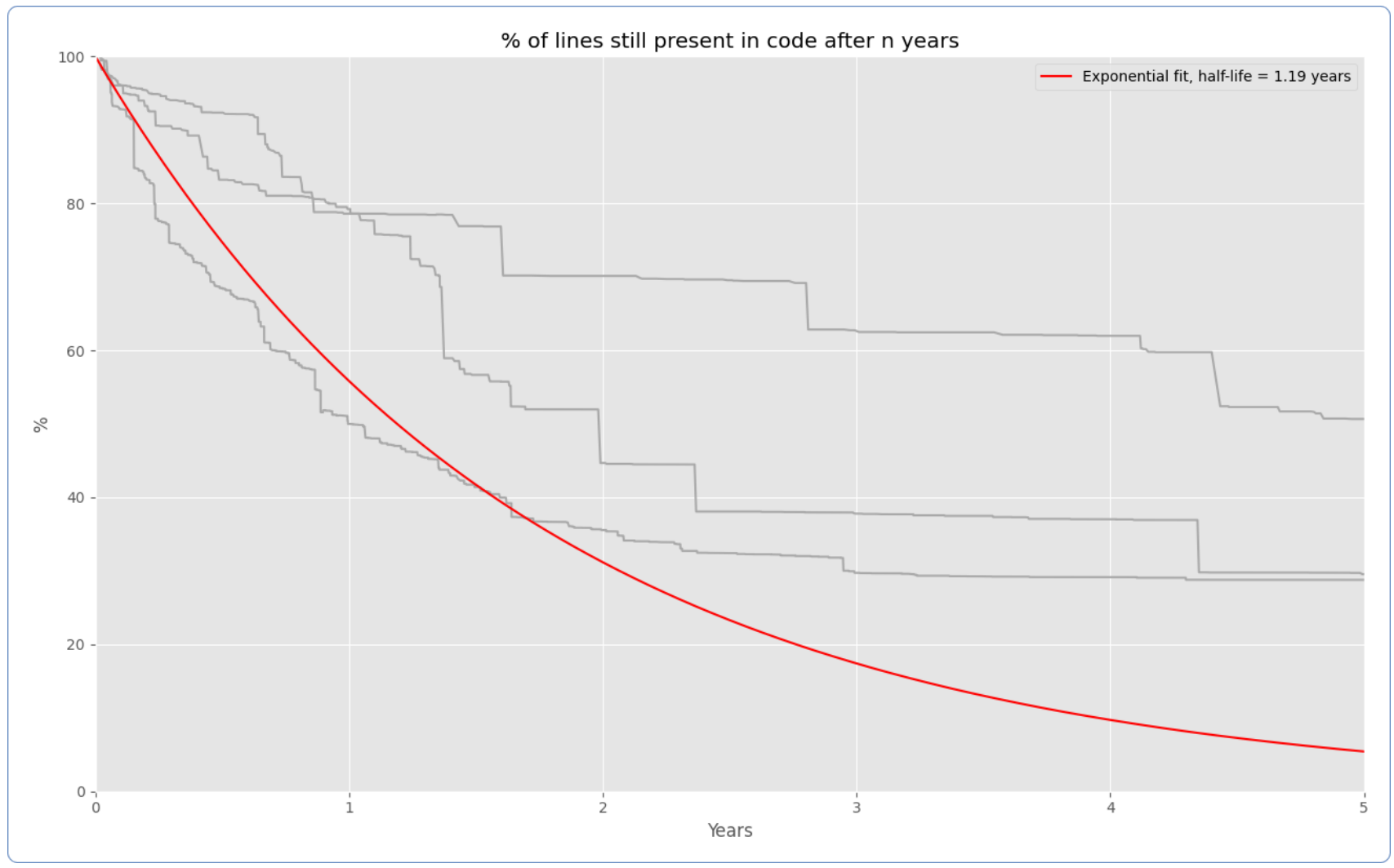


The 5-year Aggregate Decay in an Advanced Accessibility Checker plugin: Individual commits in a repo, by yearly cohorts, across 5 years

This graph (above) represents the aggregate decay and half-life of the **individual commits on the connected repos of an advanced accessibility checking plugin.** Once again, after just five years, only 25% of the original code remains.

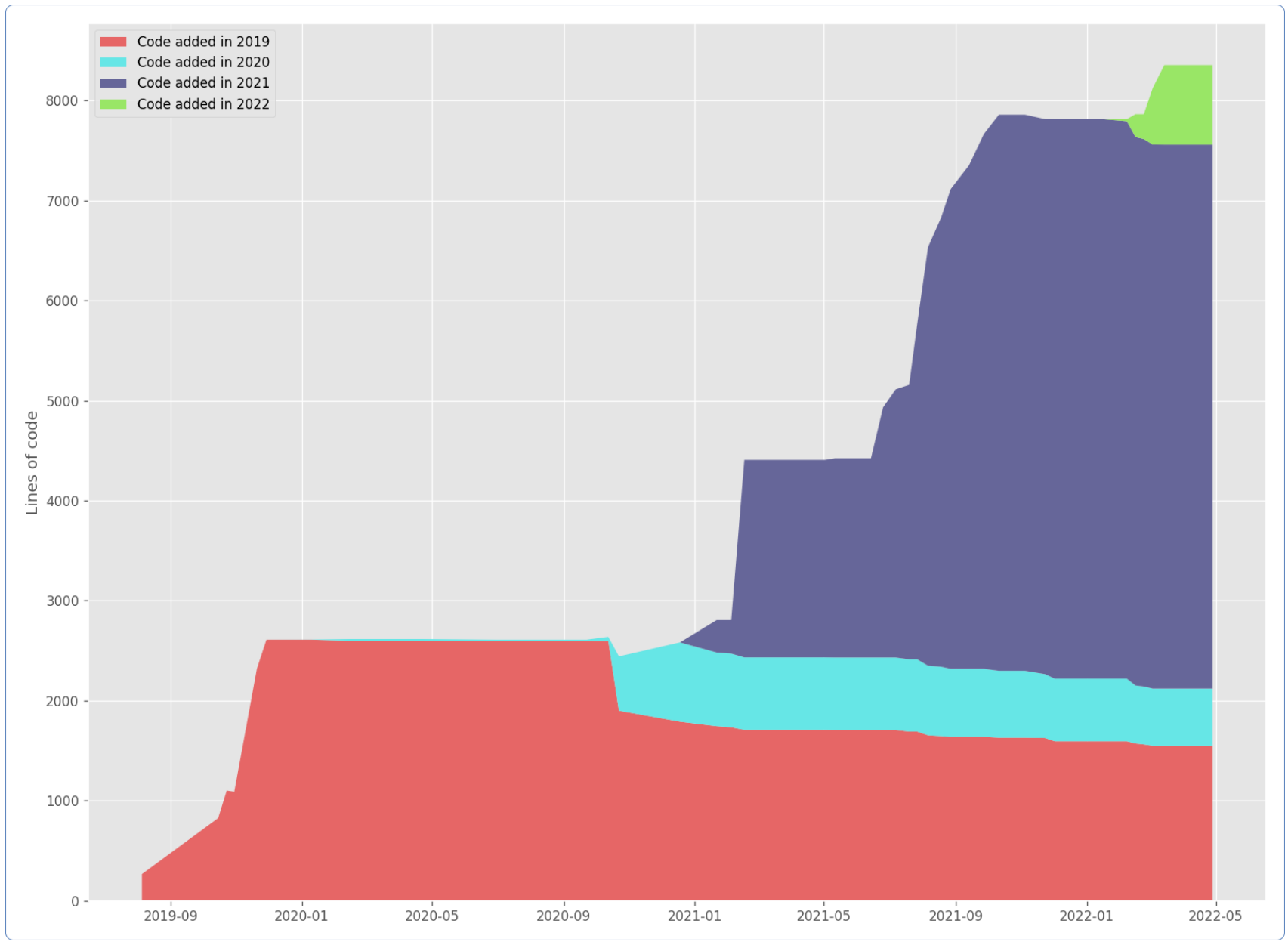
The next graph (below) represents the **decay of the connected repos of an advanced spell checking plugin,** where just under 30% of the original code is left after five years.

The 5-year Aggregate Decay in an Advanced Spell Checker plugin: Individual commits in a repo, by yearly cohorts, across 5 years

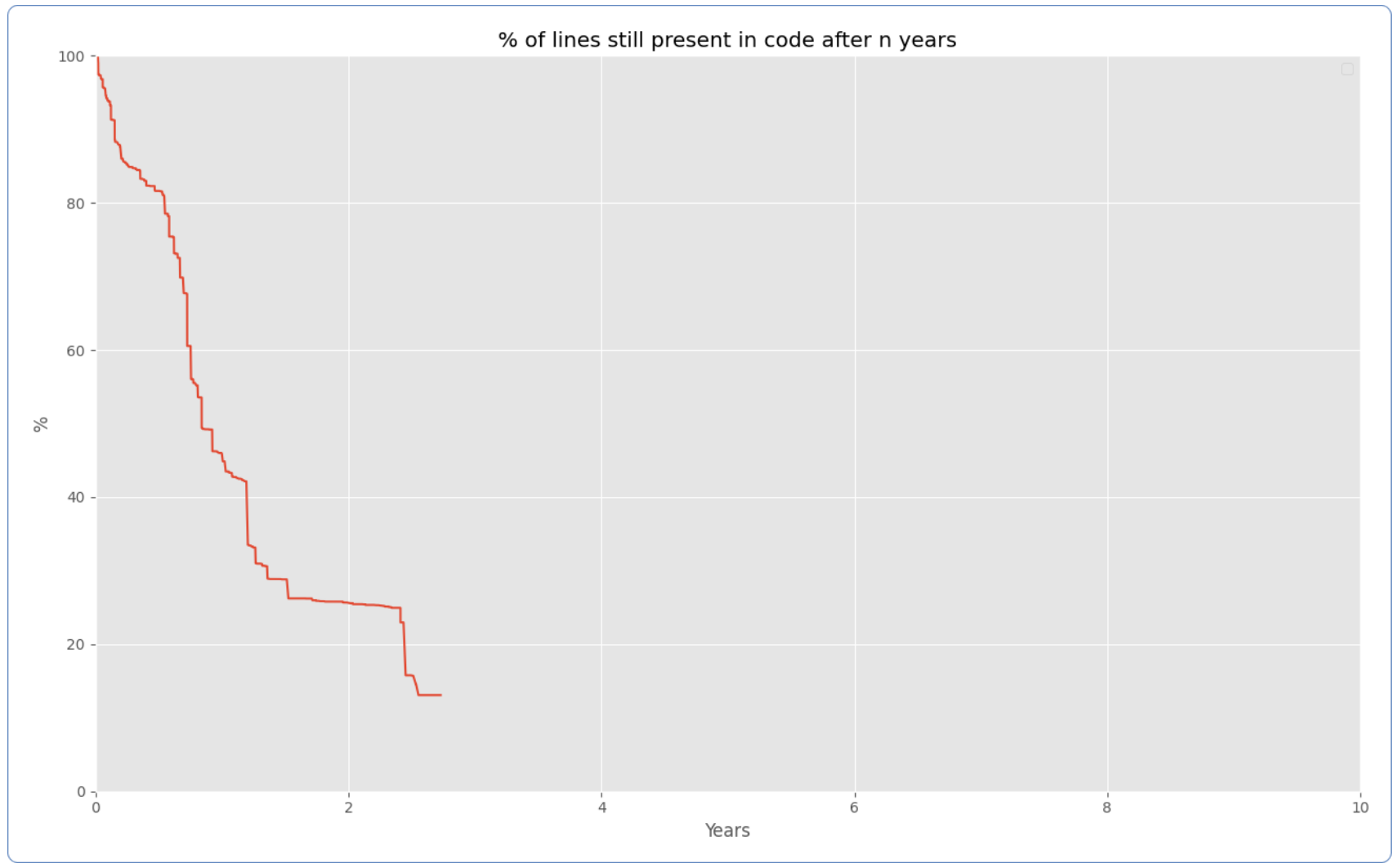


Below is the **stacked plot of an advanced tables plugin,** for a rich text editor, across just 3 years, showing the aggregate number of lines of code broken down into cohorts by the year added. In the space of just 3 years, vast sweeping changes were required to upgrade the plugin to satisfy user demands and market changes.

The 3-year Half-life of Code in a n Advanced Tables Plugins: Lines of code, by yearly cohorts, across 3 years



This final graph (below) represents the aggregate decay and half-life of the **individual commits on the connected repos of the same advanced tables plugin.** After three years of constant reworks, a mere 16% of the original code remains.



The 3-year Aggregate Decay in an Advanced Tables plugin Individual commits in a repo, by yearly cohorts, across 3 years

The key role code half-life can play when reviewing a complex code base is as an indicator of 'potential' technical debt.

While it's a given that untouched code, after enough time, probably needs work… potential tech debt is tricky. It isn't always clearly visible (until you try to read the code) and can give false impressions about the quality and quantity of the product development and/or incremental work being considered.

The maps above clearly show there's lots of 'potential' yet to be uncovered in just about every rich text editor and plugin. Which means, the payoff tasks for editors are endless. No matter how perfectly the original code was written.

Returning to Sandi Metz, "If it makes you feel any better, there's a way in which having a big mess is a sign of success. The reason your competitors don't have messes is that they went out of business. You won, and your prize is an application that betrays the ravages of time."

Shame the costs and mess, are irreconcilable.

What is half-life?

In nuclear physics, the half-life of a radioactive substance refers to the amount of time taken for it to reduce (decay) to half. Similarly, the half-life of knowledge, information or facts refers to the amount of time needed for half of the knowledge to become irrelevant or outdated.

Technical debt — also known as tech debt or code debt — is what happens when a development team speeds up the delivery of a project or functionality that will require refactoring later on. A quicker development process becomes the priority instead of high-quality code.

— **Alexandre Omeyer**
The Engineer's Complete Guide to Technical Debt, StepSize

2.5

Opportunity Cost of RTE Technical Debt

All code has technical debt. That's normal.

Big, small, start-ups and Fortune 500 companies, all carry that debt.

It's just that some applications generate more than others. So you need to be more canny.

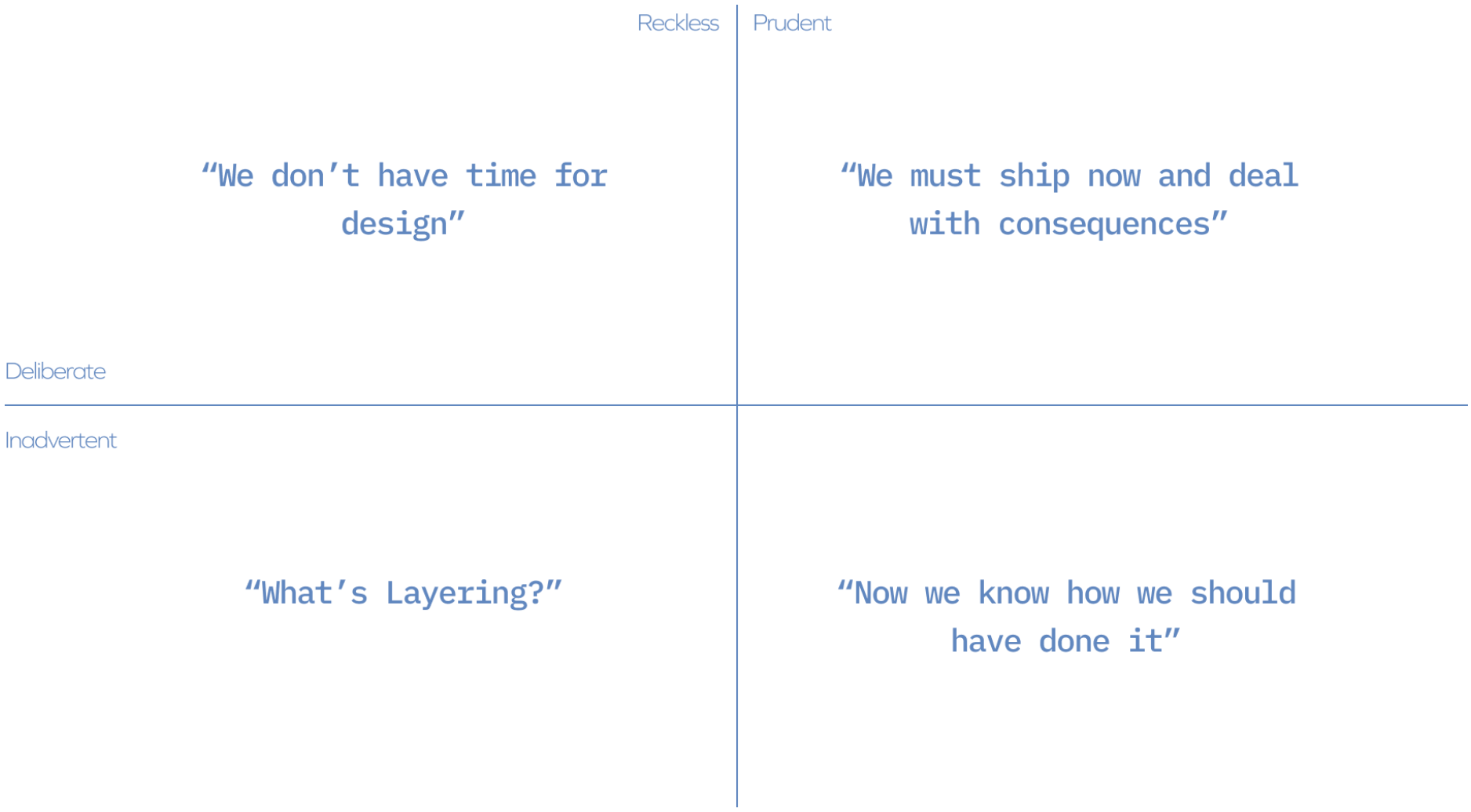
As McKinsey says, "Companies whose leadership is conversant in technology matters and has a foundational understanding of what to look for in system-architecture builds can avoid unintentionally accruing tech debt. Furthermore, involving senior management and subject-matter experts at the outset helps address critical IT considerations up front."

Technical debt should be carefully managed, to ensure its negative consequences don't exceed its advantages.

That's exactly where open source, domain experts and specialist third-party components can come into your tech stack, to help alleviate your debt burden — be the debt intentional or unintentional.

Martin Fowler usefully categorised technical debt into four quadrants, tracking along spectrums based on context and intent — from reckless to prudent and deliberate to inadvertent:

Deliberately reckless:	**"WE DON'T HAVE TIME FOR DESIGN."**
Inadvertently reckless:	**"WHAT'S LAYERING?"**
Deliberately prudent:	**"WE MUST SHIP NOW AND DEAL WITH THE CONSEQUENCES."**
Inadvertently prudent:	**"NOW WE KNOW HOW WE SHOULD HAVE DONE IT."**



Source

Each of those quadrants carry different opportunity costs and trade-offs. And seemingly small decisions can have big consequences.

The technical debt on the left-hand side of the quadrants should be avoided, at all cost. The right hand side is where the action happens.

> Prudent debt is the corollary of the agile software development principle that ‘perfect is the enemy of done’. Prudent debt can be deliberate: we know we’re going to have to fix this later. It can be inadvertent: the world has changed since we made that choice. Either way, it’s unavoidable — the objective is to minimise reckless debt and properly manage prudent debt.
>
> – 5 steps to avoid a technical debt black hole, Salesforce

Notice that the goal isn’t to reach zero debt.

Ideally, it’s a ‘Goldilocks level of tech debt’ you want — not too much, not too little. Your tech debt management plan needs to balance growth and housekeeping, to ensure you continue delivering value at high speed, without sacrificing quality.

As Hackernoon clearly puts it in their tech debt guide, " A rushed development means that the code base has certain deficiencies that a programmer will have to rework or clean up later on. These deficiencies, also called 'cruft', affect the overall code quality, and although the software can still function, it cannot reach its full potential until someone fixes the deficiencies."

But sometimes, cruft and technical debt arrives unexpectedly.

During 2021, 43 updates occurred in the Chrome browser, with 117 significant changes. That means the Chrome browser you used at the beginning of 2021 is different from the one you used at the end of the year, in at least 117 ways. Likewise, Mozilla released 10 updates over 2021, with 94 changes made to Firefox (Versions 85-94). Apple only had 3 updates for Safari (Version 15) in 2021. By any measure, all these browsers dramatically changed in a single year.

IDEALLY, IT'S A 'GOLDILOCKS LEVEL OF TECH DEBT' YOU WANT – NOT TOO MUCH, NOT TOO LITTLE.

Likewise, WordPress, the most commonly used web content management system had 6 updates with 12 significant changes and 1 major (new generation) release.

Although, these numbers aren't unique to 2021.

Every layer of a web development stack undergoes significant change each year, and the rich text editors that rely on those applications need to immediately accommodate those adjustments. There's no advance warning. It can be as simple as a security patch, or a fundamental refectoring of the code base.

You just never know.

In the past 4 years, on average there's been 8 forced fixes/patches each year, on rich text editors and their dependent plugins features, to address unexpected changes in browser behaviour.

Rich Text Editor Unplanned Fixes/Patches to Address Browser Changes

2018	2019	2020	2021
8	8	9	6

With the release of every patch, it's not just a simple code adjustment and pushing it out. Each time, testing must be done to check browser compatibility.

There are at least two desktop OS to test (Windows and OSX) and two mobile OS (Android, IOS), plus iPad, Tablet and Hybrid across multiple browsers (Chrome, Safari, Firefox/Mozilla) and screen resolutions.

Minimum Browser Compatibility Testing for Rich Text Editor Releases

DESKTOP OS		MOBILE	iPAD	TABLET	HYBRID
Windows Chrome	Windows FireFox	iOS Safari (Mobile)	iOS Safari (iPad)	Android Chrome (Tablet)	Windows Surface (Hybrid)
Windows Edge[2]	OSX Chrome	Android Chrome (Mobile)			ChromeBook (Hybrid)
OSX Safari	OSX FireFox				

All this mushrooms the Total Cost of Ownership (TCO) of a rich text editor and consumes resources: people, time and money.

Every release (Major, Minor and Patch) requires QA testing, prior to being released and depending on the priority and severity of the items within the release scope and other project risks, the launch may be delayed.

Code rewrites ensue, then back through QA for repeat testing. Every. Single. Time.

QA Testing for Rich Text Editor Releases[3]

MAJOR	MINOR	PATCH
Breaking changes (incompatible API changes) New features Improvements Bug fixes Security patches	New features Improvements Bug fixes Security patches	Bug fixes Security patches
Yearly	Quarterly	Ad-hoc
2-3 weeks (10-15 business days)	2-3 weeks (10-15 business days)	1-2 days

Out of Scope: Dev tasks related to technical debt

However if unattended, both forced and unforced changes generate tech debt.

The People Cost of Technical Debt

A recent survey by Stepsize uncovered that technical debt leads to employee churn.
More than half of the Engineers (51%) surveyed had left a company or considered leaving a company due to large amounts of technical debt, and 20% said that technical debt is the primary reason for them to leave a company.

Today's code becomes slightly out of date, very fast. Within days, sometimes. While the code may not decay (as it does with half-life), the environment in which it functions continues to change at a blistering speed.

Cost drives much of today's technology.

The typical cost of managing the technical debt generated by a rich text editor — and its advanced feature plugins — isn't small. Even discussing the possibility of adding some functionality or fixing a bug and then deciding not to do it, takes time.

And the more rework you have to do, the higher the price tag and opportunities lost.

That's the butterfly effect of technical debt decisions.

Even the Best Creates Cruft

Quality code.

It's not cruft-free: even the best carries cruft.

An accepted shorthand for many things, cruft is (generally speaking) some form of debt that eventually needs to be paid. Martin Fowler defines cruft as "Software systems are prone to the build up of **cruft** – deficiencies in internal quality that make it harder than it would ideally be to modify and extend the system further."

Sounds like technical debt.

Cruft also includes "…'things' that were left temporarily in the system during the previous iteration. Also, cruft refers to any code that is not necessary to perform the task it was designed for or forgotten code without any utility. Although cruft does not mean a code bug, it makes the code harder to maintain or to read and creates technical debt."

58%

of a developer's time is spent on code comprehension

And we all know that hard to read code, eventually equals technical debt.

A Mcrosoft study in 2017 concluded that 58% of a developer's time is spent on code comprehension, especially once the software is shipped and has to be maintained. It's something most developers intuitively know: "…the less you're familiar with the code, the longer it takes. The more surprises are in your way, the longer it takes. The harder it is to reproduce all scenarios, the longer it takes to get it right."

That's not uncommon in large, complex projects, like rich text editors, with anywhere from 484,093 to 270,122 lines of code (LOC). Different languages are used, multiple developers are coding, all with different approaches. That spells eventual tech debt.

But it also happens in single (but complex) advanced features. When a feature carries dependencies on dependencies to perform its functions, it's difficult to effectively review every single line of code, all the time.

So cruft and tech debt creeps into the code.

As the feature expands its capabilities (adding 99% accurate clean-copy-paste from Google Doc sources), or improvements are made in rushed timelines, then corners are sometimes cut and compromises made to forsake quality in favour of speed. Or there's instances when the code needs to be written so that every related app can continue, and the code quality suffers.

New discoveries also drive change — the things you didn't know about when you first set out to solve a problem for your users, like providing a 99% accurate clean-copy-paste function.

What does an advanced clean copy-paste feature do?

An advanced copy-paste plugin helps users cleanly transfer content from its source to the rich text editor (the destination). Ideally, it should automatically parse the content for security vulnerabilities, remove unnecessary style elements as well as generally clean up and modernize the background HTML.
Read more

CASE STUDY

Advanced clean-copy-paste plugin feature

TL;DR

The Word import component of an advanced copy-paste plugin (originally built in the early 2000s for a Java-based editor), was subsequently recompiled directly to JavaScript using Google Web Toolkit. By 2016 that technology was a decade old and had become unmaintainable. A rewrite was approved in 2016-17.

Triggers for the rewrite were: Large tech debt burden, poor code legibility, high maintenance, code decay.

SCOPE

A single, advanced clean-copy paste feature for a rich text editor:

- 39836 lines of code[4]
- 40 libraries
    - 18 open source inhouse libraries
    - 12 private libraries
    - 8 third-party open source libraries
    - 2 compilers (TypeScript and OCaml)

TECHNICAL DEBT/ISSUES

The original component was written in Java by an outsourced developer and looked more like C. It was poorly written and while users loved its performance, the maintenance became increasingly difficult. Particularly towards the end of its life, developers struggled to fix even minor problems – choosing to create workarounds rather than fix the component directly.

When building the new component in 2017, it was again decided not to build it in JavaScript directly, but rather use OCaml and compile to JavaScript. The resultant code was much better structured, used modern architectures and design to improve its base functionality.

However, it too has its limitations, which is why another round of rewrites is now being considered, five years later.

It seems technical debt and cruft gets the better of us all.

Even when you're the best.

Hard Choices are Opportunities

Choices need to be made.

- **INNOVATION.**
- **TECHNICAL DEBT.**
- **SPEED-TO-MARKET.**
- **OPEN SOURCE.**
- **BUY. BUILD.**
- **ASSEMBLE.**

The companies focused on rapid transformation, have already made the choice: swift innovation and continuous deployment. And they're growing. Fast.

By using specialist components.

Rapid-fire changes in customer expectations, business models and technology, are accelerating the pressure on companies to innovate. And the death and decay of countless big innovators (Blackberry, Kodak and Netscape) are constant reminders that even big, successful companies can and do disappear. Easily.

Roughly a third of new businesses exit within their first two years, and half exit within their first five years. Even the average lifespan of S&P 500 companies has dropped from 75 (1950s) to 19 years (2022) and continues to fall.

So it's innovate or be replaced. But how do you know who to trust?

In every development project, software evaluation is a critical piece of the puzzle — whether you're using open source or third-party closed components. It's a tricky balancing act between hard objectivity and subjective (but valid) individual user experience.

Here's some thought-starters. For open source components, evaluation criteria that are often used:

1. The number of developers working on the OSS
2. The number of downloads of the software
3. The developer's satisfaction
4. The level of activity on the project
5. The time between consequent releases

6. _____ The time to close bugs
7. _____ The security protocols in place
8. _____ The reputation in the community
9. _____ The quality control processes used

These are added to the characteristics you already use to measure closed source software.

Assessment Criteria for Open Source Software

According to IGI-Global, "Since the code is available to everybody it can be reviewed and assessed by using traditional methodologies that measure the level of understanding, completeness, conciseness, portability, consistency, maintainability, testability, usability, reliability, structuredness and efficiency. These assessments can be done by everybody who is interested in the quality of the OSS."

Other areas to consider when evaluating both open and closed components:

- Is the development team familiar with the technology?
- Are large enterprises using the technology (ie. Fortune 500)?
- Does the technology provide features that do not currently exist?
- Does the technology have a strong capital backing (> USD$20M) or is it backed by a global corporation (Facebook, Google, IBM, Apple)?
- At what speed is the technology growing?
- How much interest has been received from users?
- How much interest has been received from the community?
- How popular is the technology with the number of stars on GitHub (if applicable)?
- Is there a partnership that can be developed with the technology?
- Will this be an open-source project?
- Does the technology allow us to provide features that do not currently exist?
- Have sales or existing users been lost, because of the lack of this technology?
- What type of feature (business model) will this be?
- How long will it take to build an MVP of the technology?
- How long will it take to build version 1.0 of the technology?

Having an established process helps to minimize the unseen technical debt and code decay you're taking on, while still allowing you to rapidly jump-start your innovation programs. Incorporate the following seven areas into that process:

1. _____ Clear in-house process for vetting and pre-approving components (eg. Software Bill of Materials, SBoM)
2. _____ Ongoing monitoring (eg. automated visibility and control of the components)
3. _____ Reusable tech stack composed of pre-approved and vetted components (eg. cataloguing and building repositories)
4. _____ Clear licensing
5. _____ Security protocols
6. _____ QC standards
7. _____ Service agreements

By curating a repository of vetted, pre-approved components and release versions (open source, purchased and subscribed), you're minimizing security exposures, ongoing maintenance and technical debt accumulation. The components can be safely used and reused within your tech stack, across the enterprise.

That allows development to quickly move more safely, and avoids last-minute blockers during a development cycle.

Returning to where we began, with the Stripe Developer Coefficient Report, "…businesses need to better leverage their existing software engineering talent if they want to move faster, build new products, and tap into new and emerging trends."

Assembling pre-approved specialist components gives you the opportunity to safely walk the tightrope between technical debt and innovation.

And it maximizes your opportunity cost.

3.0

The Unfortunate Truth

This brings us to an unfortunate truth.

An uncomfortable truth.

That fully paying off your tech debt is highly unusual.

And undesirable.

The alternate story, within this white paper, describes a reinvention of your tech debt management — to help minimize your ‘owned’ technical debt.

That way, you sidestep making your debt hole deeper.

By tapping the domain expertise of specialists who’ve already perfected the technology you need, you avoid the trap of needlessly accruing tech debt. And the complex code and maintenance work that’s outside your strategic focus.

It reduces risk, lifts morale and delivers positive value.

By buying proven, reusable components that have built-in scalability, you’re creating reusable foundations for your future applications.

And exponential growth.

TINY TECHNOLOGIES

Tiny is the creator of **TinyMCE**, the world's most trusted WYSIWYG component that enables rich text editing capabilities within an application. Scalable, adaptable and reusable, it powers 100M+ projects worldwide and more than 1.5M+ developers use it to add velocity to their tech stacks, so they can build and ship their projects faster.

There's tens of thousands of market-leading applications powered by Tiny globally. It's helped SaaS companies, large enterprises, content creators and publishers to launch, grow and scale their businesses, reduce their development and technical debt burdens, minimize ongoing support tickets and boost the productivity of their users.